

IOS Secure Shell Support

Michael J. Martin

Last month, we looked at implementing r-command support as an alternative to virtual TTY (VTY) access to a router's EXEC command line interface (CLI). Although r-commands provide little in terms of security, their functionality as an alternative for accessing the IOS CLI make them an attractive tool for administrators of large networking environments. You may be looking for ways to massage IOS CLI output for troubleshooting and operational support. This month we start examining Secure Shell as a secure alternative for remote command execution, virtual terminal access and file transfer on IOS-based hardware platforms. This article will focus the SSH1 implementation and operation and look at potential security problems supported on IOS-based devices.

What is Secure Shell?

Secure Shell was created by Tatu Ylonen at the Helsinki University of Technology. Ylonen developed SSH while working as a researcher at the university after the network was compromised by a password-sniffing attack. It was initially conceived as means for securing remote host access, to defend against future password-sniffing attacks. SSH/SSH operates far more abstractly than an encrypted virtual terminal program, although that is its most common incarnation. SSH is a client/server application that simply provides encrypted packet mode transfer over TCP. This provides the ability to create encrypted packet-switched virtual circuits between hosts across any transmission medium, be it secure or insecure. Once a connection has been established between the client and server, anything that can utilize TCP can be transported across. This simple but dynamic approach enables SSH to provide the following services:

- Encrypted virtual terminal access (SSH): Provides remote terminal shell access on a remote server to the client host.
- Encrypted client/server file copy (scp and sftp): Exchanges ASCII and binary data between the client host and a remote server.
- Encrypted remote command execution (SSH): Client host executes commands on the remote server with execution results directed to the client host.
- Encrypted local tunneling (SSH-L): Maps a TCP service port on the client host to a TCP service port on the remote server host.
- Encrypted remote tunneling (SSH-R): Maps a TCP service port on a remote server host to a TCP service port on the client host.

SSH's Checkered Past

Early beta versions of SSH sparked wide interest. In July of 1995, SSH1 was released as a freely available client/server implementation for Unix-based systems. Later that year, as an afterthought, a draft for SSH was submitted to Internet Engineering Task Force (IETF) and SSH1 was born. Its birth was heralded by thousands of system and network administrators who benefited from a critical tool for administration that had been absent up to this point. However, all did not share in this joy, in particular the folks at RSA Security and many world governments including the United States.

The SSH1 protocol depends on two cornerstone encryption technologies that we take for granted today, but were hellfire issues in the mid 1990s. To enable secure client/server communication, the hosts must be able to encrypt the data. In order to do this, they must agree on two factors: a crypto cipher to encode the data, and a secret key to scramble/unscramble the data. Since SSH provides secure communication between hosts which may have no prior knowledge of each other, the secret key part is tricky.

IOS Secure Shell Support

Michael J. Martin

To accomplish the key exchange (needed before any data can be exchanged), SSH1 depends on the Rivest-Shamir-Adleman (RSA) asymmetric public-key algorithm. RSA is a patent on a process for public-key encryption and a cryptosystem that provides key generation, digital signatures and data encryption. Both are named after the MIT researchers who developed them. Public-key encryption works on a simple principle of two keys: one private key that is known only to the owner, and a public key that is distributed to any who wishes to send something encrypted to the issues. The asymmetry component of RSA is that for data exchange, each party must have the others public key to have a secure "exchange" of data. The problem with RSA was that in 1996 the patent was still active, requiring those that used the RSA algorithm to pay royalties to the patent holders. The RSA patent expired on September 21, 2001, and is today available for royalty free commercial and non-commercial use.

Then there was the issue of SSH1's primary data encryption method: 3DES. The Data Encryption Standard (DES) is a symmetric secret key block cipher algorithm developed by IBM in the 1970s. A block cipher operates using a single fixed size of "plaintext," which in turn is sent as a single "block" of "cipher text." DES was the U.S. government standard for non-classified data and industries such as banking. DES uses 64-bit blocks and a 56-bit encryption key. Today the DES standard is considered insecure by most individuals who are interested in keeping their data secure, though it is still widely used by many industries. Triple DES (3DES) performs three "DES" encryption passes on each 64-bit block, using two or three keys depending on the implementation. DES and 3DES are symmetric encryption algorithms because the sender and receiver both share the same secret key for data encryption and decryption.

The problem DES and 3DES is that they are both considered munitions because they have military applications. U.S. law forbids the export of cryptographic software in machine-readable form without government permission. As the need for fast, effective encryption methods increased throughout the 1990s, court challenges were made against the export restrictions. In response to these successful challenges, the U.S. government revised its export policy in January of 2000 to allow the export of "public domain" encryption code provided that proper notification is made to the U.S. Commerce Department of Bureau of Export Administration (BEA). Yet despite these major political and some real technical limitations, SSH1 was embraced and quickly deployed by the network and system administrator community.

Once fully documented, it was found that shortcomings in the original SSH1 protocol could not be corrected while maintaining backward compatibility. SSH1's technical issues -- along with those in the political and legal arenas -- brought about a radical response. After leaving the Helsinki University to tend to SSH full time, Ylonen developed a new protocol, SSH2, in conjunction with the SECSH IETF working group. The work began at Data Fellows, Ltd. (Today's F-Secure). After a falling out, Ylonen formed his own company, SSH Communications Security, Ltd. (SCS). The SSH2 draft protocol standard was released in 1996. The SSH2 product released from SCS in 1996 was a better implementation, but did not spark wide-scale migration from SSH1.

Even today, SSH1 is more widely deployed and has broader platform support than SSH2. That's partly because SSH1 was freely available as open-source, while SSH2 was initially available only as a commercial product from F-Secure or SCS, except for certain academic and non-profit uses. In 2000, SCS expanded public use to include Linux and BSD derivatives. This change in the SSH2 licensing policy, coupled with the work of Bjorn Gronvall, has driven the proliferation of a number of public SSH1 and SSH2 implementations. Of these efforts, perhaps the most widely adopted implementation of SSH was developed by Markus Friedl and the OpenBSD effort aptly name OpenSSH. OpenSSH is based on the SSH 1.2.12 OSSH code tree, supporting both SSH1 and SSH2. It is freely available for Linux, BSD, Solaris, Darwin and Windows via Cygwin.

IOS Secure Shell Support

Michael J. Martin

SSHv1 Configuration

With the exception of the latest 12.3 IOS release (and only on certain hardware platforms), the IOS's SSH support has been limited to narrow implementation of SSH1. That is to say, the IOS implementation only supports functions that are conducive to supporting router administrative functions. No remote and local tunneling here, folks. Here's what was included:

SSH1 Server for VTY access -- Introduced in IOS 12.0(5)S

SSH1 Client for the EXEC shell -- Introduced in IOS 12.1(3)T

SSH1 Server for TTY access -- Introduced in IOS 12.2(2)T

SSH1 SCP -- Introduced in IOS 12.2.(2)T (12.0(21)S on 7500 and 12000 routers)

It only included secure VTY, secure copy and support for SSH access to TTY rotary groups. And these functions were not always available in every release or feature set. The critical component for SSH support is the ability of the feature set to support DES or 3DES encryption. Serving more as a guide than a definitive list, here is a table of hardware and IOS releases that support SSH1 and SSH2 server support.

Hardware	SSH1 server/SSH1 client/SSH1 terminal-line access	SSH2 server
1x00 WAP	12.2.8(JA) - 12.2(13)JA	NA
350 WAP		NA
8xx	12.1.(1)T - 12.3(6)	NA
17xx	12.8(YN) - 12.3(6)	12.3.(4)T and later
25xx	12.1(2)b - 12.2(23) (DES only)	NA
26xx	12.1.(1)T - 12.3(6)	NA
26xx-XM	12.8(T1) - 12.3(6)	12.3.(4)T and later
36xx	12.1.(1)T - 12.3(6)	12.3.(4)T and later (3640 and 60 only)
37xx	12.8(T) - 12.3(6)	12.3.(4)T and later
71xx	12.8(T) - 12.3(6)	12.1.(19)E - 12.1.(20)E2
72xx	12.0.5(S) - 12.3.(6)	12.1.(19)E - 12.3.(4)T3
75xx	12.0.5(S) - 12.3.(6)	12.1.(19)E - 12.1.(20)E, 12.3.(4)T and later
76xx	NA	12.1.(19)E - 12.1.(20)E

As demand for SSH support has risen over the last few years, Cisco has expanded the feature sets that support it. So before going off and buying a new IOS license, check with the software advisor and see if your router's IOS version has been "enhanced" with SSH support.

Before getting into the SSH1 implementation details, I'd like to make a quick clarification. Implementing IOS SSH support is only part of the bargain. To make use of this feature, you will need a client to establish connections, as well as a server if you intend to utilize secure copy. With these facts in mind I'll cover IOS implementation along with a brief overview of OpenSSH. As I previously mentioned, OpenSSH is the most widely available free SSH1/SSH2 implementation. Most Unix/Linux distributions come with client support as part of the base install. Windows users will need to install Cygwin or use an alternative client such as putty.

IOS Secure Shell Support

Michael J. Martin

In order to implement SSH1 server support, the following must be configured on the host (router):

- A hostname
- A domain name
- One (at least) active network interface with a static IP address assignment.

Like all client/server applications, SSH1 has a client (/usr/bin/SSH) and server /usr/sbin/SSHD) component. That said, the client and server are not mutually exclusive. Hosts can have only client support, server support or both. Regardless of the service configuration, both the client and server have operational parameters that need to be set to ensure secure and proper operation. In OpenSSH environments, both the client and server components utilize a configuration file that defines their operational behavior; these files are typically located in either /etc or /usr/local/etc directories. The OpenSSH master client configuration file is called SSH_config. It contains the system-wide default parameters for client behavior, defining the types of authentication that will be accepted, the location of user identity files, protocol forwarding behavior and protocol and cipher versions supported. These parameters can be overridden if the user has their own SSH_config definitions set in their \$HOME/.SSH/ directory. The OpenSSH server configuration file is named SSHd_config. There are a fair number of server-specific attributes that can be defined; the key ones are server service port (22 is the default), version support, host key location, and (for SSH1 only) server key length and regeneration interval.

Application-specific implementations of the SSH client and SSH server, such as the one implemented in IOS, will have far fewer configurable parameters, since the implementations are limited in functionality. And those parameters that are tunable will be specific to the functional nature of the implementation. In the latest IOS release (12.3.x), there are five tunable server parameters:

Ip SSH authentication-retries (0-5)

Ip SSH source-interface (the interface from which client connections originate)

Ip SSH time-out (1-120 seconds)

Ip SSH port (2000-10000). This is used only to define SSH authentication for VTY rotary groups.

Ip SSH version (1 or 2)

Each of these are stored within the router's configuration file. The IOS client and server are limited, with support only for DES and 3DES block ciphers and password-only support for client authentication. In addition to server configuration files, SSH1 servers also have host identification keys, both public and private. The public key is sent to the client during the establishment of the connection to prove its identity. Most client implementations will store host keys for verification of future session attempts. The SSH1 server host key has a minimum length requirement of 512 bits and a maximum length of 2048 bits, with the recommended length in the middle at 1024 bits. On OpenSSH systems, the public host key is located in /etc/SSH_Host_key.pu. The private host key is located in /etc/SSH_host_key. On IOS based implementations, the public and private keys are embedded in the router's configuration. The private key is hidden, but the public key is viewable (in hex). To see the router's public key, in the EXEC shell execute the command <show crypto key mypubkey rsa>.

On OpenSSH server implementations, the host keys (public and private) are generated and installed as part of the build process. A new host key can be generated once the install process has been

IOS Secure Shell Support

Michael J. Martin

completed with the command `SSH-keygen -b 1024 -f /etc.SSH_host_key -N`. On IOS SSH1 implementations, generating the host keys is how one enables SSH server support. SSH key generation, like all IOS cryptography functions, is part of the `<crypto>` configuration command subset. Host key creation and deletion is performed within a configuration shell (the hostname and ip domain-name must be defined on the router prior to generating the host key, otherwise the process will fail) using the command `<crypto key generate rsa>`. Here is a command output example for host key creation:

```
outland-gw (config)#crypto key generate rsa
```

The name for the keys will be: outland-gw.otawa.ny.comcast.net

Choose the size of the key modulus in the range of 360 to 2048 for your General Purpose Keys. Choosing a key modulus greater than 512 may take a few minutes.

```
How many bits in the modulus [512]: 1024
% Generating 1024 bit RSA keys ...[OK]
```

```
outland-gw(config)#
Feb 16 19:22:28 EST: %SSH-5-ENABLED: SSH 1.5 has been enabled
outland-gw(config)#
```

If you need to create new host keys, you can simply run the key generation command again and the previous keys will be overwritten. To delete the host key and disable the SSH1 server, run the command `<crypto key zeroize rsa>` :

```
outland-gw(config)#crypto key zeroize rsa
% All RSA keys will be removed.
% All router certs issued using these keys will also be removed.
Do you really want to remove these keys? [yes/no]: yes
outland-gw(config)#
```

A word of caution: Running this command deletes all of the RSA keys and certificates on the router. If you are using certificate-based authentication for IPsec, you will need to reestablish the certs.

Once the host key has been generated, the SSH1 server will be enabled. (Verify that the system log message is the example above, "%SSH-5-ENABLED..."). If you want to check to make sure, the EXEC command `<show ip SSH>` will display the server version, authentication time out and permitted retries:

```
outland-gw#sh ip SSH
SSH Enabled - version 1.5
Authentication timeout: 120 secs; Authentication retries: 3
outland-gw#
```

Although the SSH1 server is enabled after key generation, the router can only be accessed with an SSH client if SSH support has been defined as part of the VTY transport input definition. This can be done by explicitly defining SSH as a transport option `<transport input SSH>` or implicitly by enabling all transport modes `<transport input all>`. The more secure option is to explicitly permit SSH. However, when enabling SSH support in pre-existing environments, be sure to provide proper notification to users and make sure there are no support requirements for other protocols before making a policy change such as this. Otherwise, you could have some upset users on your hands.

IOS Secure Shell Support

Michael J. Martin

Another factor to consider in the quest for keeping users happy is changing host keys. There will be instances where new host keys need to be generated. More commonly, the host will physically change but the hostname (and IP address) will not. Routers sometimes need to be replaced, servers are upgraded, and things change. When these types of events occur, users need to be notified. The SSH server's host key, remember, has two parts -- a private and public key. The public key is cached by SSH clients connecting to the server and stored for future verification of session. In OpenSSH implementation, these cached host keys are stored in \$HOME/.SSH/known_hosts. The IOS client does not cache host keys. When new host keys are generated or the host identity has changed and a user with a cached key opens a session, their client will receive the "new" key as essentially a phreak. In most cases, this abruptly ends the session and generates an error like this:

```
Outboy:~ martin$ SSH gw
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@      WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!      @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA1 host key has just been changed.
The fingerprint for the RSA1 key sent by the remote host is
54:46:2f:3c:e0:b1:d3:18:41:84:4a:a8:a1:5b:ec:4b.
Please contact your system administrator.
Add correct host key in /Users/martin/.SSH/known_hosts to get rid of this
message.
Offending key in /Users/martin/.SSH/known_hosts:19
RSA1 host key for gw has changed and you have requested strict checking.
Host key verification failed.
```

To correct this, the user will need to delete the invalid cached host key and connect again. Many administrators will view this kind of event as suspicious, requiring some investigation before proceeding with the session. It's a good idea if you plan to change a device's host key to inform users ahead of time.

That just about covers enabling SSH support for IOS-based hardware. Now let's get into the security issues you need to be aware of when implementing the security protocol. To understand them, you need to understand how SSH client and server authentication and communication work.

SSH Session Setup

There are almost 20 IETF RFCs covering SSH. That's a lot of reading to be sure, so to save time, [click here](#) for a nice diagram of how SSH server authentication works.

Once server authentication is complete, user authentication can begin over a secure channel.

Cisco SSH Vulnerabilities

To be fair, SSH vulnerabilities are not inherent in Cisco's implementation, but are issues with the SSH1 protocol. The vulnerabilities released in June 2001 apply to the SSH1 1.5 implementation of SSH1. They have been corrected in the SSH1 1.99 implementation, which is utilized in later IOS SSH implementations. However, the majority of IOS releases utilize SSH1 version 1.5. Administrators need to be aware of the vulnerabilities and need to assess their impact on deployment. There are three vulnerabilities:

- CRC-32 integrity check

IOS Secure Shell Support

Michael J. Martin

- Passive traffic analysis
- Server key recovery

The CRC-32 is a potential memory buffer attack. When exploited properly, an attacker can insert command code into the server or client's computer and execute it as root. The vulnerability exists when large SSH packets are received and the 32-bit packet length field is expressed as a 16-bit integer. This alteration in the expression leaves the remaining 16-bit field to be expressed as all zeros, providing the potential for code to be inserted in memory in place of the zeroed boundary. For this attack to work, the attacker needs to be somewhere along the transit path and be able to capture plain text or cipher text pairs.

The passive traffic analysis attack exploits the SSH1 protocol's use of known fixed packet lengths. Since most SSH sessions are used to exchange "text" data between hosts, there is the potential that passwords and commands could be guessed and the data could be used to assist in brute-force password attacks. This predictability is possible, because although padding is used, the actual byte length of the data without the padding is sent unencrypted as part of the SSH packet. In order to exploit the vulnerability, the attacker must have direct access to the transmission path and be able to capture the traffic.

The server key recovery attack requires the attacker to have the ability to sniff and capture traffic at the inception of the session. Once the session key has been captured, it is possible to launch a Bleichenbacher PKCS1 attack through a new server connection. Once the attacker has the server key, any data exchanged during the server key lifespan would be vulnerable.

Each of these exploits requires an attacker with "wire" access to the transmission path or server. Upgrading to IOS code versions that do not utilize the flawed code is the preferred solution. In cases where code upgrade is not an available option, administrators need to assess their degree of exposure to this type of attack.

This concludes our initial look at IOS SSH support. I hope this has been a helpful overview. Stay tuned for the next installment where we will look at implementation and use of IOS SSH.